INRIA

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau,
Rocquencourt
B.P.105 78150 Le Chesnay
France
Tél. 954 90 20

Rapports de Recherche

N° 6

ALLOCATION DE RESSOURCES DANS UN SYSTÈME RÉPARTI

Bernard MAILLOT
Alain TARABOUT
Irène VATTON

Janvier 1980

ALLOCATION DE RESSOURCES DANS UN SYSTEME REPARTI

Bernard MAILLOT, Alain TARABOUT, Irène VATTON
Laboratoire IMAG, BP 53X
38041 GRENOBLE Cedex
Tél. (76) 54 81 45

RESUME

Nous présentons d'abord un modèle de construction de systèmes informatiques par abstractions et raffinements successifs à partir d'éléments de base appelés "unités". Les unités échangent des messages au moyen d'un mécanisme de communication qui utilise un automate de transition. Ce modèle peut être utilisé pour le développement de systèmes informatiques répartis sur plusieurs machines, par exemple de systèmes de gestion de bases de données réparties.

Nous développons ensuite quelques propriétés induite par ce modèle de décomposition de systèmes sur les niveaux de représentation des données qu'il met en oeuvre.

La notion de ressource est alors introduite par référence aux notions d'unité et de machine définies dans le modèle. Nous montrons de quelle façon ce modèle peut être utilisé pour la réalisation de politiques décentralisées d'allocation de ressources.

ABSTRACT

This report deals with a model using the abstraction and refinement principles for the construction of computing systems from basic components called "units" and extension mechanisms. Unit exchange messages through a general communication mechanism using transition automata. This model can be used for the development of systems which have to be distributed on several physical machines, for example a distributed data base management system.

We then focus on some properties induced by the model on the representation level of the data in a system constructed according to its principles.

Ressources are introduced by reference to the notions of units and machines defined in the model. It is shown how the model can be used for the implementation of distributed ressource allocation mechanisms.

SOMMAIRE

1. Préliminaires : systèmes et machines

D'une manière générale, un système informatique (et en particulier un système de gestion de base de données) peut être considéré sous deux aspects principaux :

- comme l'interpréteur d'un langage particulier, grâce auquel sont masquées certaines contraintes du matériel. Le langage définit alors une machine abstraite plus facile à manipuler que le matériel brut.
- comme un allocateur de ressources, gérant le matériel pour le compte des utilisateurs de manière aussi fiable et efficace que possible.

Il offre, à travers le langage de la machine abstraite qu'il définit des fonctions d'accès aux ressources physiques qu'il gère en les répartissant entre ses utilisateurs. Ces fonctions d'accès apparaissent comme des objets typés [LIS] : les ressources logiques mises en oeuvre par la machine abstraite.

Il y a alors analogie entre les ressources physiques et le langage d'une machine "matérielle" d'une part et les ressources logiques et le langage d'une machine abstraite d'autre part. On peut donc définir des systèmes informatiques en termes d'autres systèmes, c'est-à-dire construire des machines abstraites sur d'autres machines (abstraites ou physiques).

C'est par l'opération d'abstraction [DAH] que l'on définit une machine abstraite, c'est-à-dire que l'on spécifie les ressources logiques offertes en interface. Construire une machine abstraite en termes d'autres machines c'est effectuer une opération de raffinement [WIR]. Le terme de machine désignera dans la suite une machine abstraite d'un niveau quelconque, aussi bien qu'une machine réelle.

On va donc construire des système par niveaux de machines édifiées les unes sur les autres (on dit encore par inclusions les unes dans les autres). Les ressources logiques fournies par les machines des divers niveaux sont définies comme des réalisations de modèles d'objets, munis des actions portant sur ces objets (les fonctions d'accès). Il est donc

naturel de structurer les programmes qui réalisent un niveau de machine en modules, chaque module étant défini [PAR] comme l'association d'objets et de fonctions manipulant ces objets.

Considérons un usager soumettant un programme à une certaine machine M, c'est-à-dire une suite d'instructions du langage interprété par M. L'exécution de chaque instruction du programme donne lieu à l'invocation d'une ou de plusieurs fonctions d'accès localisées dans les modules de la machine. Les fonctions d'accès sont elles-mêmes des programmes qui sont interprétés par des machines de niveau inférieur.

Les relations entre machines de niveaux différents sont des relations d'interprétation.

2. Présentation du modèle

2.1. Processus communiquant par messages

Nous utilisons la méthode d'abstraction et de raffinements successifs pour construire des niveaux de machines abstraites composées de modules [MAI1].

Deux schémas d'éxécution différents peuvent être définis sur une telle structure; ils diffèrent essentiellement par l'emploi de la notion de processus.

Le premier schéma d'exécution est celui de l'appel procédural : on considère que toutes les fonctions qui interviennent dans un programme utilisateur donné s'exécutent sous l'autorité d'un même processus utilisateur, avec appels successifs et empilement des contextes d'exécution propres à chaque module dans une pile associée au processus, et dépilement au retour. Ce schéma possède plusieurs inconvénients :

- perte de place en mémoire du fait que les piles d'exécution des processus contiennent des contextes entiers de modules, ce qui est le plus souvent inutile [BRI] ;
- pas de possibilité d'exécution "pipe-line" ou parallèle pour un même utilisateur [JAM, LAU] ;
- difficulté de répartir un système fonctionnant suivant ce schéma sur des processeurs distincts, et en particulier de remplacer un ou plusieurs modules par un composant matériel (microprocesseur avec mémoire,

par exemple).

Ces inconvénients disparaissent dans le deuxième schéma d'exécution, qui est celui que nous avons adopté : en effet nous avons considéré que chaque module est le support de processus cycliques, chacun étant chargé d'une fonction du module. Les appels intermodules sont alors réalisés par transmission de messages entre processus cycliques.

Ce schéma, qui est celui de HYDRA [WUL], permet de considérer qu'un processus se comporte comme producteur et consommateur de messages. L'aspect dynamique d'un système fonctionnant suivant de tels principes est celui d'un réseau de processus communiquant par messages.

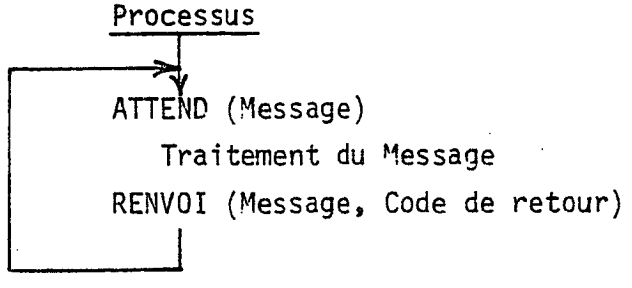
Ce schéma s'écarte de celui de HOARE [HOA1] en ce que dans notre modèle les messages sont tamponnés et non échangés sur rendez-vous, et en ce que nos processus sont cycliques et ne sont donc pas créés à l'arrivée d'un message dans un module ni détruits à la fin du traitement dans le module.

Dans notre modèle, la mise en oeuvre du mécanisme de communication nécessite deux primitives :

ATTEND : le processus qui exécute cette primitive se met en attente d'un message.

RENVOI : le processus qui exécute cette primitive renvoie le message qu'il vient de traiter.

De façon schématique, un processus exécute une boucle sur un programme comprenant un appel d'ATTEND et un appel de RENVOI.

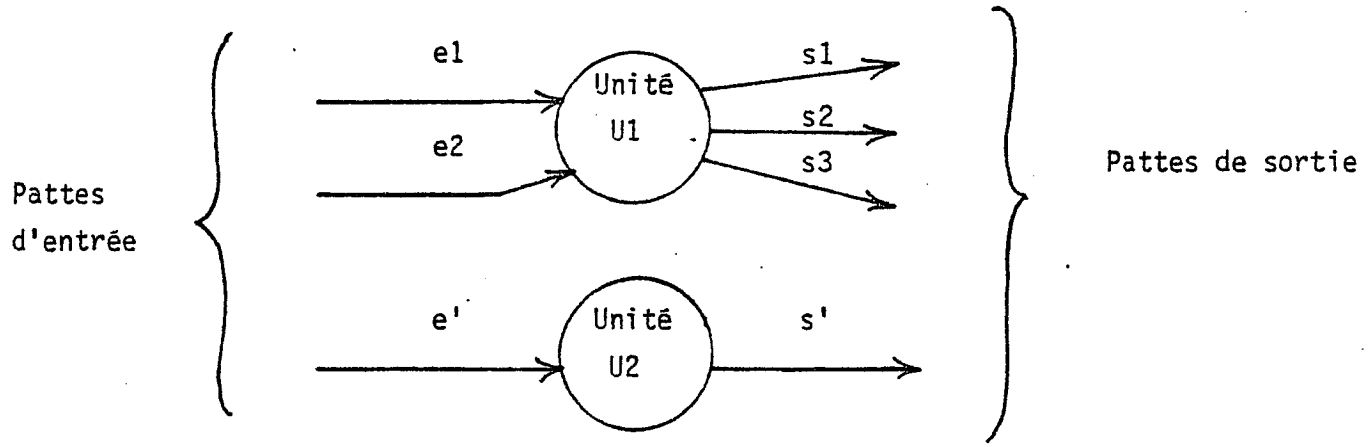


Ces programmes, rappelons le, sont contenus dans des modules. Nous donnons le nom d'unité à ces modules communiquant entre eux par le truchement des processus qu'ils supportent. La notion d'unité couvre donc à la fois la notion statique de module et la notion dynamique de processus.

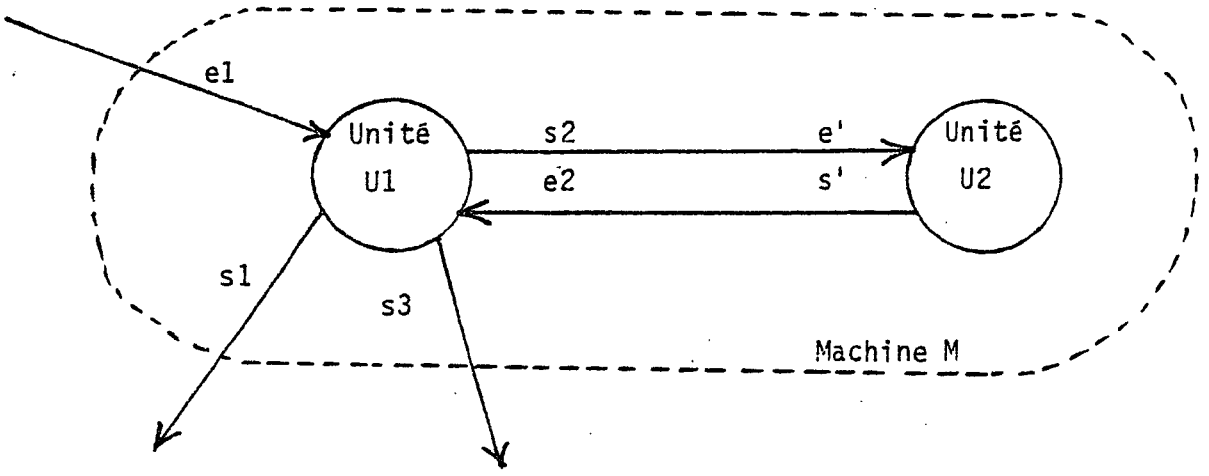
2.2. Connexion d'unités

Fonctionnellement, une unité se présente comme une boîte noire avec :

- des pattes d'entrée, chacune correspondant à une fonction d'accès au module;
- des pattes de sortie correspondant à divers codes de retour des fonctions d'accès du module.



Les unités qui composent une machine sont connectées entre elles en associant des pattes de sortie avec des pattes d'entrée d'unités différentes selon un certain graphe de connexion défini de manière statique. Au moment du RENVOI effectué par une unité, l'acheminement du message est déterminé par le graphe de connexion suivi.



Ces graphes de connexion sont réalisés sous la forme d'automates d'états finis que nous appelons automates de transition et qui déterminent le champ des transitions valides entre unités d'une même machine.

Cette méthode apporte les avantages suivants :

- les connexions peuvent être définies indépendamment de la programmation des unités;
- une unité ne dépend aucunement des unités auxquelles elle est connectée;
- il n'y a pas de limitation au nombre de connexions à une unité (toutefois une telle limitation peut exister pour les unités matérielles);
- il est possible de réaliser des appels procéduraux entre unités.

2.3. Structuration en machines hiérarchisées

Une machine est construite à partir d'unités en définissant un automate de transition. Fonctionnellement, une machine se présente alors exactement comme une unité, avec :

- des pattes d'entrée qui forment un sous-ensemble des pattes d'entrée de ses composants,
- des pattes de sortie qui forment un sous-ensemble des pattes de sortie de ses composants.

Une machine peut être utilisée exactement comme une unité pour construire une nouvelle machine. En d'autres termes, une machine peut être constituée d'unités et de machines incluses.

Lorsqu'un message arrive sur une patte d'entrée d'une machine incluse, l'identification de l'automate de la machine englobante est empilée dans une pile associée au message, en même temps que l'état courant de cet automate. L'automate de la machine incluse est alors utilisé pour l'acheminement du message.

Lorsqu'un message sort d'une machine par une de ses pattes de sortie, les opérations inverses des précédentes sont exécutées.

Nous avons vu (cf. § 2.1) que dans chaque unité le mécanisme de communication de messages était sollicité par les instructions ATTEND et RENVOI.

Nous avons créé deux unités ATTEND et RENVOI pour interpréter les instructions du même nom : en prédéfinissant ces unités dans une architecture nous donnons la possibilité de créer des machines contenant ces unités.

En d'autres termes, ces machines définissent des langages contenant les instructions ATTEND et RENVOI. Les processus exécutés par ces machines peuvent donc recevoir, traiter et renvoyer des messages; ils peuvent donc servir à définir de nouvelles unités. Et ainsi de suite...

Finalement, on peut définir par extensions successives des machines à partir d'unités et de machines incluses préalablement définies, pour peu que les machines des différents niveaux intermédiaires contiennent les unités ATTEND et RENVOI que nous avons prédéfinies.

2.4. Implémentation

Les mécanismes de construction sont réalisés, dans notre modèle, au moyen des trois primitives suivantes :

- CREER-PROCESSUS (machine abstraite, programme).

Un nouveau processus est créé, correspondant à l'exécution du programme par la machine abstraite.

- CREER-UNITE (machine abstraite,
  programme,
  nombre de processus cycliques,
  nombre d'entrées,
  nombre de sorties).

Une nouvelle unité est créée par regroupement de plusieurs processus sur un même programme et une même machine abstraite.

- CREER-MACHINE (composants,
  connexions,
  entrées,
  sorties).

Une nouvelle machine est créée par assemblage d'unités et/ou machines déjà existantes. Les connexions entre les entrées et les sorties de ces différents composants sont définies par un automate de transition.

L'acheminement effectif des messages entre les unités est réalisé par l'intermédiaire des primitives ATTEND et RENVOI déjà citées.

Ces cinq primitives sont mises en oeuvre dans une machine abstraite élémentaire que nous appelons noyau [MAIL], implanté directement sur le matériel.

2.5. Construction de systèmes répartis

Une unité est construite sur une machine - abstraite ou physique - pouvant interpréter les instructions ATTEND et RENVOI. Ces instructions sont aussi bien câblées que microprogrammées ou interprétées par des unités logicielles. D'une manière générale toute unité de notre modèle peut être réalisée par câblage , microprogramme ou programme.

Il est possible, par assemblage de telles unités, de construire des systèmes répartis sur des architectures multiprocesseurs, chaque composant étant un processeur aussi bien matériel que logiciel.

On peut distinguer deux sortes de processeurs [MAZ]:
- les processeurs spécialisés correspondant aux unités de notre modèle;
- les processeurs banalisés correspondant aux machine incluses.

Par conséquent, notre modèle peut prendre en compte différentes sortes d'architectures décentralisées : ordinateurs centraux avec satellites, multi-microprocesseurs, machines-réseau. Il permet tous les degrés possibles de décentralisation du contrôle des communications entre les composants puisque les unités ATTEND et RENVOI peuvent être décentralisées de toutes sortes de manières.

Nous appelons site un ensemble de machines pour lesquelles le contrôle de l'acheminement des messages est réalisé par le même couple d'unités ATTEND et RENVOI.

Les automates de transition utilisés pour la connexion des unités d'un site donné sont construits par une unité spécialisée, appelée connecteur qui interprète l'instruction CREER-MACHINE. Un connecteur particulier peut être implanté sur chaque site, mais il est également nécessaire de disposer d'un connecteur commun à tous les sites puisque ceux-ci sont connectés par l'intermédiaire d'un automate de transition, que nous appelons "global", et des couples d'unités ATTEND/RENVOI des différents sites : chacune de ces unités doit recevoir une copie de l'automate global lorsque les connexions sont définies.

Les informations de contrôles associées à un automate global font partie des données du message circulant dans la machine répartie et sont interprétées dans les unités ATTEND et RENVOI de chacun des sites, lesquelles font transiter le message entre les composants de chaque site selon le sous-automate local jusqu'à ce qu'une transition finale du sous-automate local provoque le renvoi du message vers un autre site selon l'automate global.

Un automate local peut lui-même inclure des sous-automates si certains composants du site sont des machines incluses.

Nous avons présenté dans [MAI2] un exemple de conception d'un mini-SGBD réparti sur une architecture multiprocesseur qui est conforme au modèle que nous venons de présenter.

3. Niveaux de machines et de représentation des données

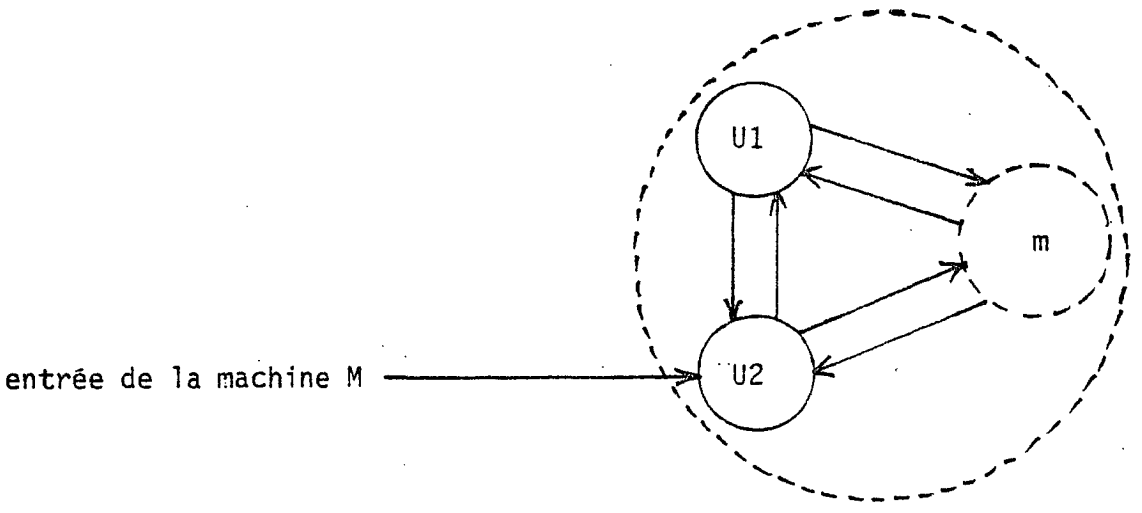
Chaque machine abstraite définit un répertoire d'instructions et d'objets qui constitue le langage de programmation "machine" de cette machine abstraite (cf. ch. 2).

Dans ce chapitre, nous essaierons de montrer que la méthode d'abstraction-raffinement induit certaines relations entre les langages et les objets définis à chaque niveau de machine. Le cas échéant, nous suggérerons un certain nombre de règles d'utilisation des outils qui sont proposés dans notre modèle.

3.1. Enrichissement et réduction des langages

Ce sont les unités d'une machine qui interprètent les instructions du langage défini par cette machine. Donc si l'on veut modifier le langage d'une machine, il faut soit modifier certaines unités, soit ajouter ou enlever des unités.

Cette dernière solution peut être mise en oeuvre facilement sans que l'on ait besoin de modifier l'automate définissant la machine initiale. Il suffit en effet de construire une nouvelle machine constituée des nouvelles unités et de la machine initiale [MAI2].



m : machine initiale définissant un langage l
U1 : unité interprétant une nouvelle instruction i
U2 : unité filtrant les appels à m.

La machine M définit alors un langage L qui, par rapport à l, est enrichi de l'instruction i. Mais à l'inverse, le filtre effectué par U2 a pour effet que la puissance du langage L est réduite par rapport à celle du langage l.

3.2. Niveaux d'interprétation

Lorsqu'on crée un processus sur une machine, on fournit un message initial qui définit un programme. Ce programme est ensuite interprété par la machine au fur et à mesure que le message chemine entre les unités.

Lorsqu'on crée une unité sur une machine abstraite, on fournit un message initial qui définit le programme que tous les processus de l'unité exécuteront. Au moyen des unité ATTEND et RENVOI, ce programme, qui est interprété, devient lui-même un interpréteur, ou plus exactement une partie d'interpréteur si l'on considère que c'est chaque machine abstraite qui est l'interpréteur du langage qu'elle définit.

Les unités ATTEND et RENVOI jouent un rôle essentiel dans la programmation d'un interpréteur car elles constituent le moyen par lequel cet interpréteur peut recevoir des commandes et envoyer des résultats.

Les unités ATTEND et RENVOI sont disponibles à n'importe quel niveau de machine. En conséquence, ces instructions peuvent être intégrées dans n'importe quel langage sans qu'on doive se soucier de les interpréter. Il en résulte en particulier qu'un interpréteur ne se met jamais en attente d'une commande pour le compte des programmes qu'il a interprété, ce qui lui permet d'être disponible au mieux pour ces programmes.

Toutefois, si l'on n'a pas besoin d'interpréter les instructions d'attente de commande et d'envoi de résultats, il faut quand même définir pour ces instructions une syntaxe qui s'accorde avec le langage dans lequel elles sont intégrées.

3.3. Niveaux de représentation des données

Le schéma fondamental du traitement de l'information est le suivant [HOA] :
- lecture d'une information
- traitement de l'information
- écriture de l'information traitée.

Ce schéma est manifestement récursif, et chaque étape de la récursion est généralement associée dans une application à une couche (ou niveau) de logiciel ou de matériel. Dans le modèle de décomposition que nous avons présenté, chacune de ces couches correspond à un niveau de machine abstraite et nous avons développé dans ce chapitre la relation d'interprétation qui relie les programmes exécutés sur chaque niveau de machine. Mais à chaque niveau de machine correspond également un niveau de représentation des données manipulées par ces programmes, que ces données soient dans les mémoires des machines ou bien qu'elles soient échangées avec le monde extérieur.

Dans cette relation, un objet défini sur un niveau de machine i sera représenté par une structure d'objets définis au niveau i-1.

Une instruction de traitement de l'objet de niveau i sera alors interprétée sur des machines i-1 au moyen d'instructions de lecture, d'écriture et aussi de traitement d'objets de niveau i-1, conformément au schéma indiqué plus haut.

On notera cependant qu'il est d'usage courant d'offrir simultanément à l'utilisateur plusieurs niveaux de représentation et d'accès aux objets. Par exemple, dans le cas des fichiers :
- le niveau fichier : les primitives d'accès sont OUVRIR et FERMER
- le niveau enregistrement : les primitives d'accès sont LIRE et ECRIRE
- le niveau caractère : les primitives d'accès sont les instructions de manipulation de chaînes de caractères.

Cela correspond au fait qu'il est pratiquement impossible de définir à chaque niveau un ensemble de primitives suffisamment puissant qui permette de faire tous les traitements possibles sur les objets du niveau. On préfère donc laisser ce travail à l'utilisateur en lui fournissant pour chaque objet du niveau i, sa représentation en termes d'objets de niveau i-1, ainsi que les primitives d'accès à ces objets.

Dans notre modèle, cette solution se traduit par l'inclusion de la machine de niveau i-1 dans celle de niveau i.

Un certain nombre de précautions doivent cependant être prises dans l'utilisation de cette possibilité d'inclusion et plus généralement dans toute mise en oeuvre, avec notre modèle, de niveaux d'accès aux données.

a) En premier lieu, nous dirons qu'une relation d'inclusion entre des machines ne peut remplacer une relation d'interprétation que dans la mesure où un assemblage quelconque d'objets d'un certain niveau constitue toujours une représentation correcte d'un objet du niveau supérieur. Dans la pratique, ces deux relations coexistent souvent, c'est-à-dire que, si une machine A contient une machine B, elle contiendra également un certain nombre d'unités construites sur B. De cette façon, les programmes déroulés dans cette unité pourront travailler sur le niveau B de représentation des objets de A; ils pourront en particulier vérifier que les assemblages d'objets de B fabriqués par l'utilisateur sur la machine B forment des représentations correctes d'objets de A. C'est ce que fait par exemple une méthode d'accès direct pour un fichier en format fixe :

- au niveau de l'enregistrement, elle vérifie que le nombre de caractères fournis est correct,
- au niveau du fichier, elle vérifie que l'index fourni avec l'enregistrement se situe bien dans les limites définies par la taille du fichier.

b) La deuxième remarque concerne les problèmes de synchronisation qui se posent lorsque des objets de A sont partagés par plusieurs processus utilisateurs, lesquels ont également accès au niveau B de représentation

des objets de A. Il appartient au constructeur d'une machine ou d'une unité de gérer correctement l'accès aux objets qu'elle permet de manipuler, puisqu'en effet l'existence même de ces objets n'est pas connue du noyau. Suggérons cependant deux principes qui devraient permettre de faciliter cette gestion :

- premièrement faire en sorte que dans l'automate associé à chaque machine, une transition corresponde toujours à un état de cohérence des objets manipulables sur cette machine. Concrètement, lorsque plusieurs unités doivent collaborer à la construction d'un objet, nous suggérons donc d'isoler le sous-automate associé à cette collaboration en définissant une machine incluse.

En respectant ce principe, l'expression de la synchronisation peut alors être faite au niveau des constituants d'une machine. Remarquons d'ailleurs que sur une machine réelle, les opérations de base sont en général ininterruptibles, ce qui garantit la cohérence des objets manipulés.

- Deuxièmement, lorsque les utilisateurs ont la possibilité d'accéder directement à plusieurs niveaux de représentation des objets d'une machine, il est prudent de ne leur fournir à chaque niveau que des copies d'objets du niveau supérieur. L'utilisateur ne risque alors pas de détériorer les objets puisqu'il ne travaille que sur des copies; mais copies et originaux se situent bien sûr au même niveau de représentation. Les chaines d'objets qu'un utilisateur échange avec une méthode d'accès fichier sont un exemple caractéristique de copies d'objets, les objets en question étant des enregistrements de fichier.

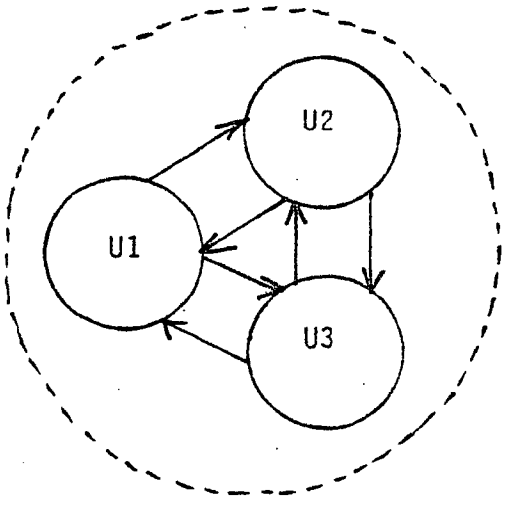
L'utilisation des copies n'est cependant pas obligatoire, ce qui permet d'éviter une multiplication coûteuse de ces copies lorsqu'on déroule une cascade d'interprétations.

Par exemple, dans le cas cité plus haut d'une méthode d'accès direct, des copies ne sont créées qu'au niveau de l'enregistrement. Mais on notera que la cohérence du fichier ne peut alors être garantie qu'en bloquant l'accès au fichier pendant tout le temps qu'un processus y accède en

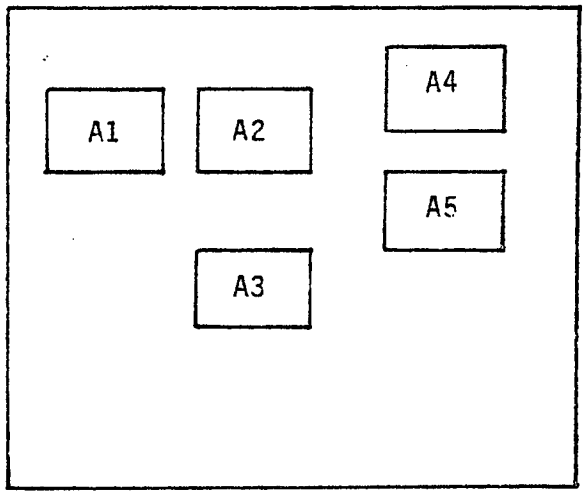
écriture. Disons pour résumer qu'on a le choix entre deux méthodes : d'une part créer des copies de façon à limiter les conflits d'accès à la mémoire de la machine abstraite, ce qui permet de gérer ces conflits assez simplement; d'autre part ne pas créer de copies, mais gérer assez finement (sur chaque objet) les accès à la mémoire de la machine si l'on ne veut pas que celle-ci constitue un goulot d'étranglement.

3.4. Niveaux de désignation des objets

La mémoire d'une machine abstraite contient les objets accessibles au niveau de cette machine; l'adresse de chaque objet dans la mémoire constitue sa désignation.



machine A



mémoire de A

Soit donc la machine A dont la mémoire est constituée des objets créés ou préexistants sur cette machine, soient A1, A2, A3, A4 et A5.

Chaque objet est créé par l'une des unités U1, U2 ou U3 (ou par plusieurs) sur demande d'un processus s'exécutant sur la machine A. Cet objet reçoit en même temps de la part de ces unités une adresse qui doit permettre de le retrouver dans la mémoire de A. Nous appellerons nom unique de l'objet cette adresse.

Supposons maintenant que les unités U1, U2 et U3 sont construites sur une machine B. Les programmes déroulés par ces unités manipulent donc exclusivement des objets de B. Ce sont ces programmes qui déterminent la représentation des objets de A en termes d'objets de B. Le mécanisme de désignation des objets décrits pour A s'applique à toutes les machines, de sorte que la représentation de chaque objet de A se compose des noms uniques des objets de B qui le constituent, et d'un schéma de montage interprétable par la machine A.

Il se pose alors le problème de savoir où doivent être conservées ces représentations. Il n'est guère possible de les conserver systématiquement dans la désignation : la désignation de chaque objet contiendrait alors les désignations des objets qui le composent, et ainsi de suite jusqu'aux désignations des objets de base, qui sont les octets.

Une autre solution consiste alors à gérer dans chaque machine un catalogue des objets créés par cette machine. Ce catalogue permet d'accéder à la représentation de chaque objet au moyen de son nom unique. Il est géré par les unités de cette machine et constitué lui-même d'objets de niveau inférieur.

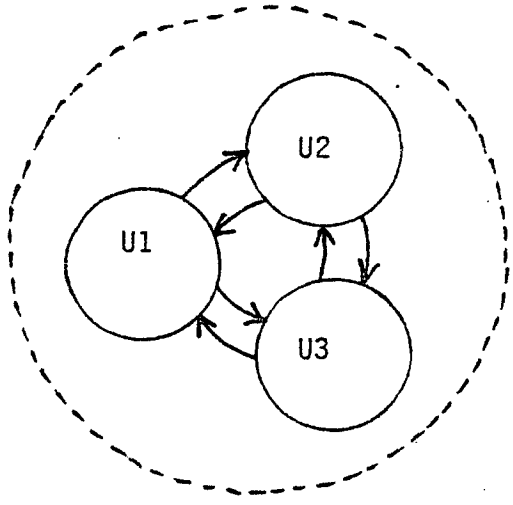
Généralement la machine donnera au catalogue qu'elle gère la même représentation qu'aux objets qu'elle définit.

Considérons par exemple un système de gestion de fichiers. Il définit une machine abstraite dont les objets sont les fichiers. Chaque fichier possède un nom unique et est constitué d'un certain nombre de pistes. Le catalogue des fichiers est lui-même un fichier qui contient donc son propre nom. Mais le système de gestion de fichier doit évidemment pouvoir accéder au catalogue sans passer par son nom. Les numéros de pistes qui portent le fichier catalogue doivent donc figurer en constantes dans les programmes du système de gestion de fichiers.

Ajoutons que le schéma de désignation que nous avons suggéré s'applique uniquement aux originaux des objets et non aux copies. En effet, celles-ci ne sont jamais accessibles au niveau supérieur même si elles ont la même représentation que les originaux. Elles n'ont pas de nom unique et ne sont donc pas cataloguées.

Dans le cas des enregistrements d'un fichier par exemple, un ordre d'écriture devra avoir pour paramètre la copie d'un enregistrement, mais aussi le nom unique de l'enregistrement dont cette copie doit remplacer la valeur.

Nous dirons pour terminer que les deux méthodes de désignation suggérées coexistent le plus souvent dans un système. C'est notamment le cas lorsqu'un niveau de machine B n'est utilisé que comme support d'un niveau de machine A.



Exemple :

Les unités U1, U2 et U3 sont construites sur la machine B qui ne supporte pas d'autres unités ou processus. Si l'on suppose que les unités de la machine B ont été construites sur une machine C, alors il n'est pas besoin de gérer un catalogue dans la machine B si le catalogue de A contient pour chaque objet de A les noms uniques des objets de C qui le constituent. En d'autres termes, le catalogue de la machine B est remonté dans celui de A.

Cela entraîne les deux conséquences suivantes sur la désignation des objets de B :

- les unités U1, U2 et U3 désignent les objets de B par leur représentation complète en termes d'objets de C.
- tout objet de B est constituant d'un objet de A. Si la machine B est incluse dans A, les programmes déroulés sur la machine A ne peuvent donc accéder qu'à des objets de A à leurs constituants de niveau B. Ils désignent alors ces derniers par une notation pointée : nom unique d'un objet de A . nom local d'un constituant de niveau B.

Dans le cas où une seule copie est maintenue pour chaque objet, le partage d'un objet peut entraîner le partage de la copie. Il est alors nécessaire de mémoriser dans le catalogue de la machine abstraite la constitution des copies en même temps que celle des objets originaux : à chaque constituant d'un objet on associe le constituant correspondant dans la copie. La notation pointée indiquée ci-dessus désigne alors les constituants de la copie et non pas ceux de l'original puisque ces derniers ne sont pas accessibles.

3.5. Le partage des données. Problèmes de synchronisation

L'outil de communication de base que nous avons défini est le message. Les informations échangées entre les unités transitent donc dans les messages. Mais la taille d'un message est forcément bornée supérieurement.

Lorsque les informations à transmettre sont volumineuses on a parfois recours à la technique de fragmentation, comme dans les réseaux d'ordinateurs, mais l'échange des informations reste une opération coûteuse. Aussi préfère-t-on limiter, lorsque c'est possible, les transferts d'informations. La communication s'effectue alors conjointement par accès à des mémoires partagées et par messages.

Dans ce paragraphe, nous allons nous intéresser au rôle particulier joué par chacune de ces deux méthodes dans la communication. En particulier

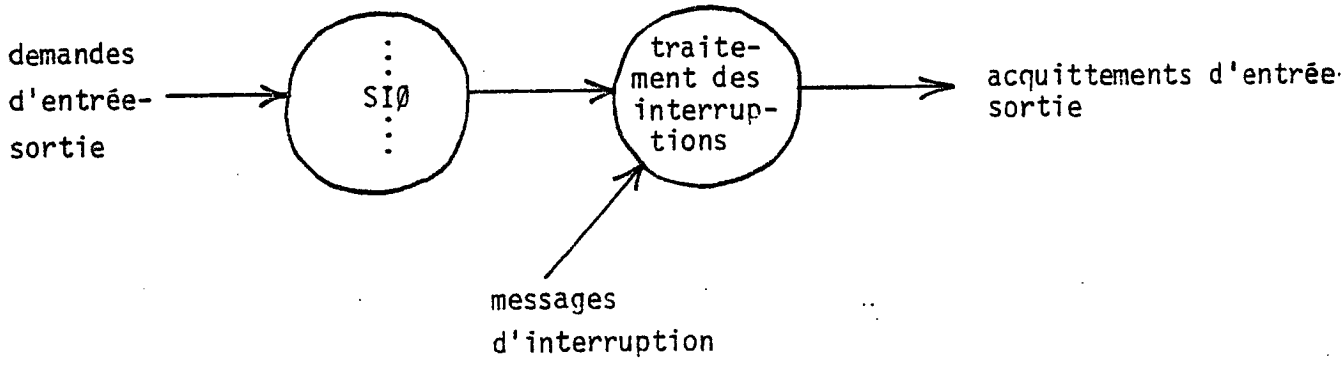
nous distinguerons le partage de données considéré comme moyen de communication du partage de données qui résulte d'activités intrinsèquement parallèles (ex : base de données partagée par n utilisateurs).

Nous aborderons ensuite brièvement les problèmes de synchronisation résultant du partage de données.

a) Le partage des données

Il y a une forme de partage qui correspond à une nécessité logique : ainsi un même objet d'une base de données peut être accédé par plusieurs processus via plusieurs contextes d'accès.

Il existe une autre forme de partage qui permet d'éviter des transferts d'information entre unités : prenons le cas d'un superviseur d'entrée/sortie contenant une unité chargée de lancer les entrées-sorties par l'instruction SIØ et une autre unité qui reçoit les messages associés aux interruptions d'entrée-sortie.



Ces deux unités ont en commun les tables décrivant l'état de chaque périphérique. Ces tables ne sont pas transmises dans les messages et ces derniers ne transportent que des adresses dans ces tables.

De façon générale, lorsque plusieurs unités accèdent à des données communes, les messages qu'elles échangent contiennent des références plutôt que des valeurs.

Les deux schémas suivants se substituent alors l'un à l'autre.

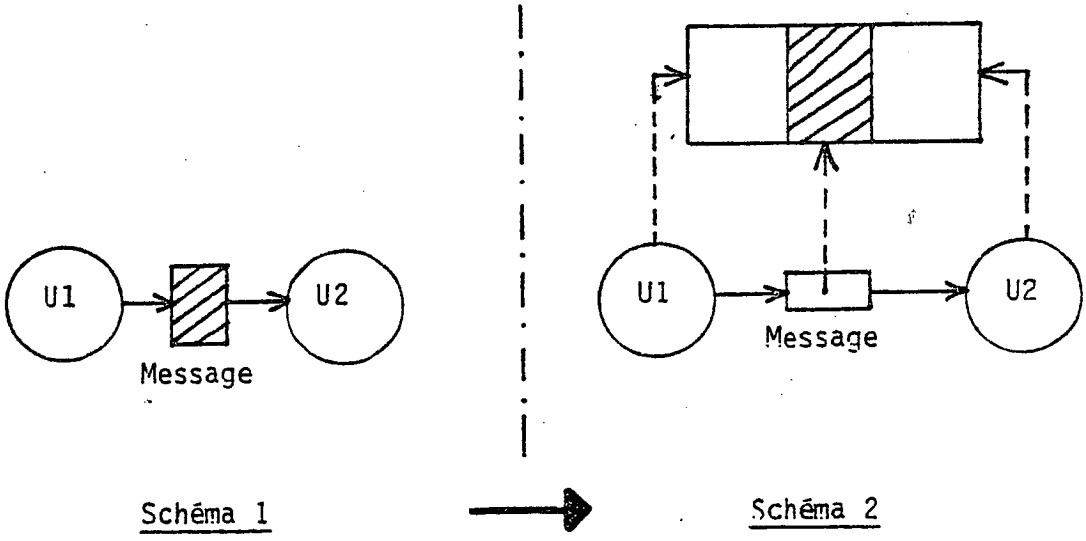


Schéma 1 → Schéma 2

Les flèches en pointillé représentent l'accès à une mémoire commune

b) Synchronisation

Les deux formes de partage que nous venons d'illustrer ont cependant un point commun : elles sont source de conflits au niveau de l'accès aux données et ces conflits ne peuvent être résolus que s'il existe une section critique commune à tous les chemins d'accès à chaque donnée partagée.

Dans l'exemple précédent du superviseur d'entrée/sortie, cette section critique commune est la fonction d'accès aux tables décrivant les périphériques.

Nous retrouvons ici la notion de module qui est un ensemble de données munies des fonctions d'accès à ces données. Un module peut correspondre à une unité ou à un groupe d'unités.

Une section critique commune ne doit pas obligatoirement apparaître au niveau même où l'on programme l'accès. Par exemple, des instructions machine de lecture et d'écriture en mémoire centrale peuvent apparemment être exécutées en parallèle lorsque plusieurs processeurs partagent la mémoire. Il y a cependant une section critique au niveau de l'accès à la mémoire de sorte que des instructions d'écriture dans un même mot ne peuvent pas être exécutées en même temps et ne riquent donc pas d'interférer.

Notre point de vue est que la section critique est l'outil de base à partir duquel des instructions de synchronisation plus évolués peuvent être définies (sémaphores, moniteurs,...).

Le modèle de décomposition permet de réaliser cet outil de base et, qui plus est, cet outil peut être réalisé à n'importe quel niveau de machine.

En effet, à n'importe quel niveau de machine, il est possible de créer une unité n'ayant qu'un seul processus serveur. Cette unité représente alors automatiquement une section critique pout tout accès aux données qu'elle manipule.

On peut ainsi programmer des unités moniteurs [HOA2] à tout niveau.

La communication par messages permet donc de synchroniser des processus sur des données partagées.

4. Outils d'allocation fournis par le modèle

Nous abordons ici les problèmes d'allocation de ressources et notre propos est de montrer que l'outil de construction de systèmes que nous avons défini est en même temps un outil pour l'allocation de ressources.

Nous allons essayer de situer la notion de ressource par rapport aux notions d'unités et de machines définies dans notre modèle de décomposition. Ce faisant, nous montrons de quelle façon les unités et les machines peuvent être utilisées pour la mise en oeuvre de politiques décentralisées d'allocation de ressources.

4.1. La notion de ressource

L'architecture d'une application étant constituée de machines, d'unités et de processus, il paraît intéressant de considérer que les ressources consommables par un élément quelconque de cette architecture sont les constituants de la machine sur laquelle cet élément est construit.

Deux schémas d'allocation sont alors envisageables :

a) La prise en compte d'un message par une unité représente l'allocation d'une ressource au processus qui est associé au message, et le renvoi du message représente la libération de la ressource. Sur une machine réelle par exemple, le chargement du contexte d'un processus sur l'unité centrale représente l'allocation de cette unité centrale au processus.

b)La ressource doit rester allouée au processus pendant plusieurs appels successifs. Des opérations d'allocation et de libération sont alors définies en plus des opérations d'accès par l'unité qui gère la ressource. Les périphériques sont gérés en général de cette façon dans un superviseur, le schéma standard d'utilisation d'un périphérique étant le suivant :

- ALLOUER-PERIPH (nom symbolique du périphérique désiré)
- TRANSFERER (nom unique du périphérique, paramètres) n fois
- LIBERER-PERIPH (nom unique du périphérique).

L'allocation d'une ressource se traduit alors pas le RENVOI d'un message sur une patte de sortie particulière de l'unité qui gère cette ressource.

Quel que soit le schéma d'allocation utilisé, il n'appartient pas aux unités qui gèrent ou qui représentent des ressources de contrôler cette

allocation, c'est-à-dire de choisir parmi les demandes lesquelles elles vont prendre en compte; elles n'en ont d'ailleurs pas les moyens. La seule politique qu'elles puissent appliquer dans ce domaine est celle dite du "premier arrivé, premier servi".

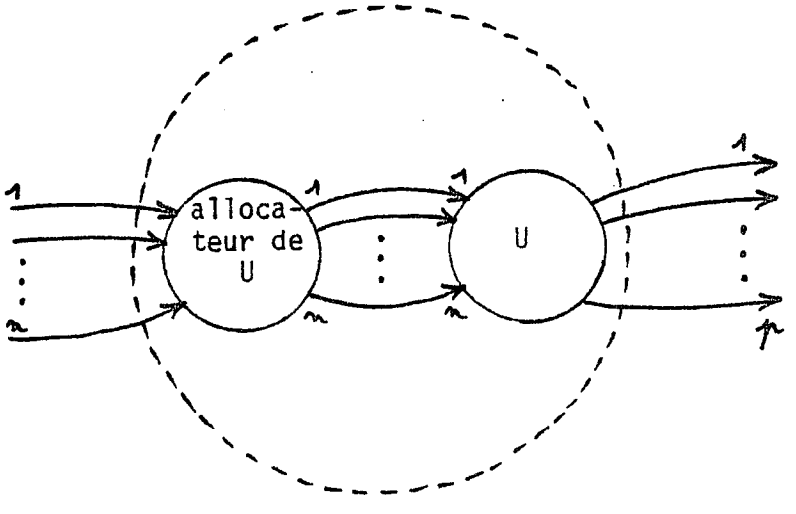
Si l'on veut pouvoir contrôler les avancement relatifs des processus dans une architecture, il est donc nécessaire de fournir des outils permettant de définir pour chaque ressource une politique d'allocation de cette ressource entre les demandeurs qui se présentent.

Nous allons voir que la communication par messages et le modèle de décomposition que nous avons présentés fournissent potentiellement ces outils.

4.2. Les allocateurs de ressources

L'exécution d'un processus par une machine se traduit par le cheminement d'un message entre ses unités. Il est donc facile de contrôler les allocations et libérations de ressources.en filtrant les messages avant qu'ils parviennent aux unités.

La solution la plus triviale consiste alors à remplacer chaque unité par une machine constituée de cette unité et d'une unité que nous appelons "allocateur" et dont le rôle est de filtrer toutes les demandes.



machine U

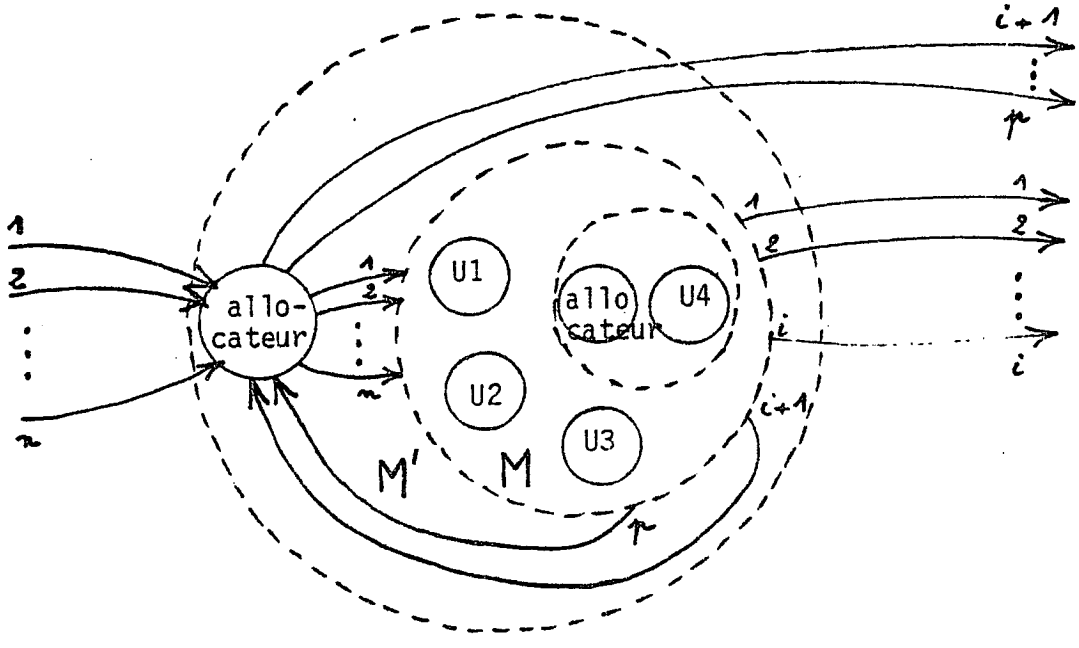
L'allocateur de U est l'unité d'entrée de la machine U; les pattes d'entrée et de sortie de la machine U sont les mêmes que celles de l'unité U.

Rien n'empêche que plusieurs unités possèdent le même allocateur, si par exemple elles collaborent à la gestion d'une même ressource, mais on aura quand même autant de machines "allouées" que d'unités si l'on se limite à cette solution triviale.

Une solution plus économique consiste à définir une unité d'allocation non pas pour chaque ressource, mais pour chaque groupe de ressources. On constate d'ailleurs que les ressources d'un système ne sont pas toujours indépendantes et que la définition d'une politique d'allocation doit être faite à un niveau relativement globale

Le mécanisme que nous proposons se résume en définitive à associer un allocateur à chaque machine abstraite et à placer cet allocateur aux "endroits-clés" dans l'automate de transition de la machine.

L'utilisation de ce mécanisme conduit à la définition de machines ainsi constituées :



Machine M' = (M "allouée")

L'allocateur doit être l'unité d'entrée de la machine M' et la machine M ne doit jamais apparaître seul dans l'architecture, mais toujours englobée dans M', faute de quoi des processus pourraient s'exécuter sur M en échappant à son allocateur.

L'allocateur peut en même temps être intégré à la machine M selon le schéma proposé au début. Cela correspond au fait que la politique d'allocation même si elle est définie globalement au niveau d'une machine doit pouvoir être mise en oeuvre plus finement au niveau de certains composants.

4.3. Quelques stratégies d'allocation

Il appartient au constructeur d'une machine de définir les tâches de l'allocateur de sa machine et de le programmer en conséquence.

Nous donnons ici à titre d'exemple quelques stratégies applicables au niveau d'un allocateur.

a) La transmission d'un message est différée ou refusée lorsque l'unité destinataire est surchargée, la surcharge étant mesurée par le nombre de messages que l'unité n'a pas encore renvoyés.

b) Des priorités sont affectées aux processus supportés par une machine donnée.

c) Avant de transmettre un message, l'allocateur y insère des commandes ou des paramètres pour l'unité destinataire. Prenons l'exemple d'une unité capable d'abandonner un traitement en cas de dépassement d'une limite de consommation (nombre de lignes imprimées, durée d'exécution,..) passée en paramètre dans le message. L'allocateur peut fixer la valeur de ces paramètres ou bien délivrer la valeur demandée par tranches successivrs pour effectuer un multiplexage de l'unité.

d) L'allocateur collecte des statistiques sur l'emploi des ressources ou sur les flux des messages.

e) Dans le cas où le multiplexage des unités entraîne un risque d'interblocage, l'allocateur exécute un algorithme de prévention. La définition d'un

outil centralisé de synchronisation tel que celui décrit dans([MAI1] § III. 1.22) peut faciliter grandement la prévention des interblocages.

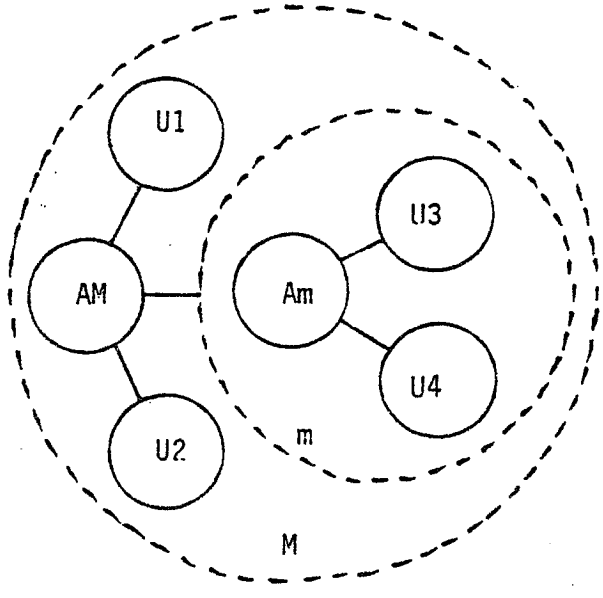
Les attributions des allocateurs sont donc potentiellement très variées. Elles dépendent essentiellement des applications supportées.

4.4. Allocation décentralisée

Le mécanisme que nous avons suggéré permet de décentraliser la fonction d'allocation de ressources puisqu'un allocateur différent peut être défini pour chaque machine abstraite.

Une politique d'allocation peut donc être définie localement pour chaque machine. Cette possibilité est voisine de celle qui est offerte dans le projet HYDRA par les "Policy Modules" [LEV].

Il faut remarque que ces niveaux d'allocation sont asservis de manière hiérarchique :



AM : Allocateur de la machine M
Am : allocateur de la machine m.

La machine M est composée des unités AM, U1 et U2 et de la machine m. Si l'on suppose que les ressources de la machine m sont allouées par Am entre tous les processus qui se présentent, alors la politique d'allocation

fixée par Am ne saurait être influencée par AM. Cependant, l'ensemble des ressources de m constitue une ressource dans la machine M et une politique d'allocation de cette ressource peut être mise en oeuvre par AM.

5. Conclusion

Nous pensonsavoir montré que le modèle de décomposition de systèmes informatiques que nous proposons contient des mécanismes universels qui permettent

- de construire des systèmes par abstractions et raffinements successifs;
- de réaliser de tels systèmes sur des architectures réparties;
- de définir des niveaux de langage par enrichissement et réduction de langages préexistants;
- de structurer les données traitées par niveaux de représentation;
- d'exprimer des mécanismes de désignation et d'accès par niveaux à ces données;
- de définir des politiques d'allocation, propres à chaque niveaux de machines, des ressources fournies par cette machine.

Remerciements

Ce modèle a été appliqué à la réalisation d'un système expérimental [MAI1], ce qui en montre la validité. L'équipe OURS (Outils de Recherche en Systèmes) de l'IMAG qui a réalisé ce système se composait des personnes suivantes : J. Briat, J. Estublier, P. Laforgue, B. Maillot, J. Raymond, S. Rouveyrol, X. Rousset de Pina, A. Tarabout, I. Vatton.

6. Bibliographie

(BRI) J. BRIAT, J.P. VERJUS
Implication de certaines propriétés d'un noyau de système (MAS) sur un langage d'écriture..
Bulletin BIGRE n°4, octobre 1976.

(DAH) O.J. DAHL, E.W. DIJKSTRA, C.A.R. HOARE
"Structured programming", Academic Press, 1972

(HOA1) C.A.R. HOARE
Communicating sequential processes
CACM, Vol. 21, 8, août 1978

(HOA2) C.A.R. HOARE
Monitors : an operating system structuring concept
CACM, Vol. 17, 10, octobre 1974

(JAM) A.J. JAMMEL, H.G. STIEGLER
"Structural decomposition and distributed systems"
Leibniz Rechenzentrum, Munich, novembre 1977

(LAU) H.C. LAUER, R.M. NEEDHAM
On the duality of operating system structures.
2ème colloque international sur les systèmes d'exploitation, IRIA, octobre 1978

(LEV) R. LEVIN & al.
Policy/mechanisme separation in HYDRA
Proceedings 5th symposium on operating systems principles, Austin, novembre 1975.

(LIS) B. LISKOV, S. ZILLES
"Programming with abstract data type"
SIGPLAN notices, Vol. 9, 4, 1974

(MAI1) B. MAILLOT, A. TARABOUT, I. VATTON
"Un outil de recherche en systèmes informatiques"
Thèses docteur ingénieur et 3ème cycle, INPG, décembre 1978

(MAI2) B. MAILLOT, A. TARABOUT, I. VATTON
Application du principe d'abstraction-raffinement à la conception de systèmes répartis
Séminaire sur les systèmes répartis à partage de données,
Aix-en-Provence, Colloques IRIA, mai 1979

(MAZ) G. MAZARE
"Systèmes multi-microprocesseurs : problèmes de parallélisme, définition et évaluation d'un système particulier"
Thèse d'Etat, Grenoble, juin 1978

(PAR) D.L. PARANAS
"On the criteria to be used in decomposing systems into modules"
CACM, Vol. 15, n° 12, 1972.

(WIR) N. WIRTH
"Program development by stepwise refinement"
CACM, Vol. 14, n° 4, 1971

(WUL) W.A. WULF editor
"The HYDRA operating system"
Carnegie Mellon University.